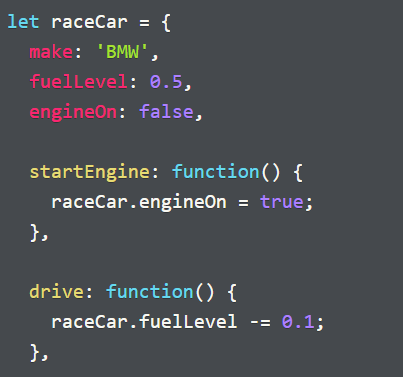
# Lesson 1

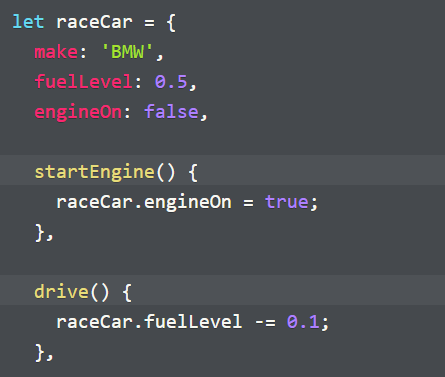
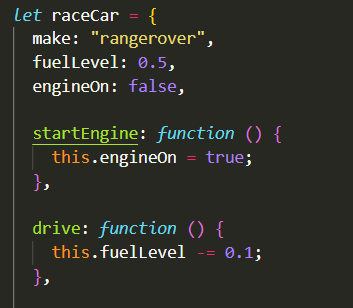
## What is OOP?

1. A style of programming that uses objects to organize a program.
2. Instead of thinking about a program in terms of series of steps (like in procedural programming), you instead think in terms of a collection of objects that interact with each other.
3. These objects have states (or properties) and behaviors. Ex: the object car has states like color, no. of doors mileage etc. and behaviors like it can be started, reversed, parked etc. We have some data and operations that use and manipulate that data.  
   
4. Since programs written in OOP way are well organized by their very nature, they are easier to reason about and maintain (since code has fewer external dependencies).
5. OOP code is flexible, easier to understand as well as modify.
6. Cons:
   1. The code is relatively larger in size
   2. Can often be less efficient, thus requiring more disk space and computing power.

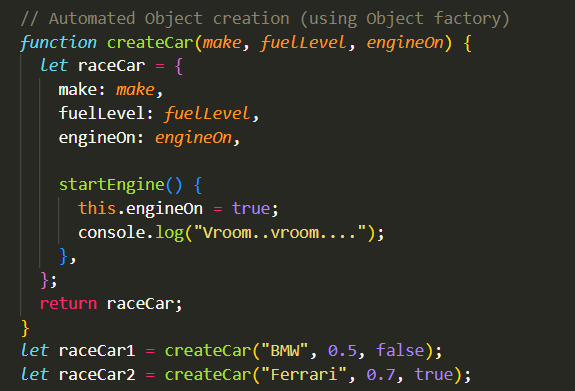
## Encapsulation

1. The idea of bundling and combining the data (state) and the operations (behavior) that work on that data in a single entity, ex: An object.
2. The idea of combining data and functionality (operations relevant to that data) into a single functional whole in OOP is called encapsulation.
3. Encapsulation also refers to restricting access to states and certain behaviors, only exposing data and behavior that other parts of the application need to interact and work with.
4. That is, the object exposes a public interface to work with and keeps all other implementation details hidden. Other objects cannot change the data of this object without going through a proper interface.
5. Earlier, JS did not have access restrictions but since ECMAScript 2022 JS supports access restriction using a feature called “**private class fields**”.
6. **Methods:** When object properties are functions, we call them methods. Methods are normally in charge of changing the state of the object.

## OOP

1. **Compact syntax:** A shorthand syntax for using functions as object values (methods). You do not need colon and keyword ‘function’.   
   
2. **this** keyword:
   1. keyword that references the object (name) from inside the object.
   2. Suppose we change the variable name (raceCar) or pass the object to a function that uses a different name for its argument, then using the original name to reference the object will throw a reference error.
3. **Collaborator objects or collaborators:** 
   1. Objects that are used as a state in another object, it is named after the idea of objects working together in some form.
   2. Collaborators play an important role in OODesign, they represent the connections between different classes in your program.
   3. While working on OOprograms, it is important to understand what collaborators your object needs and whether those associations make sense from the standpoint of modeling the problem your program aims to solve.
   4. Collaborators do not always have to be custom objects, they can be built-in objects also like arrays, dates etc.

**Note:** Objects collaborate with other objects by using them as part of their state. We say that two objects have a collaborator relationship if one object is part of the state of the other.

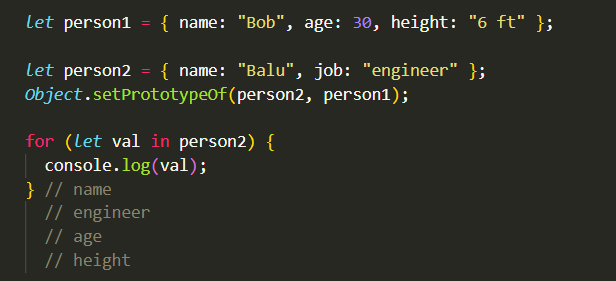
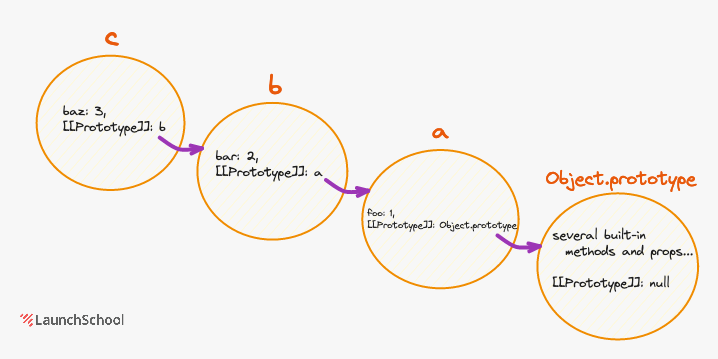
1. **Object factories**:
   1. Functions that create and return objects of a particular type.   
      
   2. Object factory functions provide an abstraction for object creation.

## OO Rock-Paper-Scissor

1. Approach for planning an OO application:
   1. Write a textual description of the problem or exercise
   2. Extract the significant nouns and verbs from the description.
   3. Nouns are the objects/type of objects and verbs are the behaviors/methods. Organize and associate the verbs with the nouns.
2. Here, we do not think of the game flow logic in the early phase. Initially, all we do is organize and modularize the code into a cohesive structure- objects.
3. Once you know what all objects you need (along with their data and behavior), you can look at orchestrating the program’s flow.
4. **Inheritance using factory functions:** call parent objectin the current factory fn, merge the parent object with the current object using **Object.assign**(parentObj, currentObj).

# Lesson 2

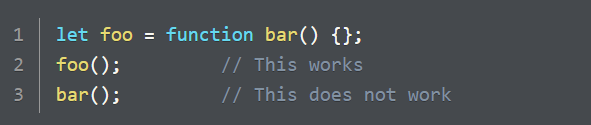
## Object Prototypes

1. JS uses **prototypal inheritance.** The object that you inherit properties and methods from is called the **prototype**.
2. We can create a prototype by using: obj2 = **Object.create**(obj1). Here, all the properties and methods of obj1 are inherited into a new object obj2, hence obj1 is the **prototype object**.
3. Methods **hasOwnProperty**() and **Object.hasOwn**() both check for the property only within the object and return false if not found. ‘**in’** operator on the other hand checks for the property in prototype object as well.
4. JS objects use an internal [[prototype]] property to keep track of their prototype. You can use this property to access prototype properties:
   1. Object.getPrototypeOf(obj2)
   2. Object.setPrototypeOf(obj2, obj1) (need to create empty object first: obj2 = {}). These two lines together are equivalent to obj2 = Object.create(obj1)
5. Also, the [[prototype]] property stores the reference to the prototype object hence if there is any change in the prototype object, then those changes are seen in the inheriting object as well.
6. All the objects have the function Object.prototype as their default prototype
7. You can iterate over all the keys of an object using for/in. Note: this will also loop over all the properties in its prototype as well.  
   
8. If we want only the properties defined within the object itself, we can use the method Object.**hasOwnProperty**() as a filter and then log the property.
9. The static methods of Object: keys(), values() and entries() also return only the object’s own properties/values in an array.
10. Both for/in and keys deal only with enumerable properties of the object (objects that can be iterated over). Most properties/methods that are built in are not enumerable, while the ones that you define are enumerable.
11. Use the method Object.prototype.**propertyIsEnumerable**() to check whether a property is enumerable.
12. **The prototype chain**:
    1. Since prototype of an object is also an object, it can also have its own prototype (further up as a predecessor), this is called a prototype chain.  
       
    2. An ex. of prototype chain is given above. b is the prototype of c and a is the prototype of b. Further, the prototype of a is the default prototype; Object.prototype. The prototype of this default prototype is null.
    3. You can also use objName.\_\_proto\_\_ (AKA dunder proto) to access the prototype object, it is deprecated but you may come across it in legacy code bases.
    4. When you try to access a property of an object, the look up for that property occurs according to the prototype chain until it reaches Object.prototype, if the property is not found there also, then the property access evaluates to undefined.
13. It is possible to create objects whose prototype chain does not end with Object.prototype.
    1. All you have to do is set its prototype property to null. Such objects are called **bare/clean objects**. They are generally used for key/value data structure and do not carry the baggage of unneeded properties and prototypes.  
       
    2. Bare objects do not have access to methods/properties like hasOwnProperty(), toString() etc.

## Function expressions

1. We know that functions defined using function declaration can be executed even before the function definition code. However, function expression can only be executed once it is defined.
2. This is because JS runs our code in two passes, in the first pass it does some preparatory work (like hoisting) and in the second pass it executes the code.
3. During hoisting, the engine moves the function declarations to:
   1. the top of the program file in which they are defined or
   2. on the top of the function in which they are nested.

Note: Hoisting isn’t limited to function declarations.

1. The function name given to a function expression is not visible from the scope where the function expression is defined.  
   
2. Type of functions: The type of functions (uninvoked ofcourse) is ‘function’. What kind of value is a function? It is an object (not a primitive). ‘function’ are a compound type that have their own properties and methods.
3. Higher Order Functions:
   1. Functions that take another function as an argument and/or return another function (like any other return value).
   2. Using HOF you can create functions that abstract away repeating patterns in a class of functions. For ex: Array.prototype.map()
      1. Declares/initializes a result array
      2. iterates over the current array
      3. performs some mapping operation on each element of the array
      4. pushes that to the result array
      5. return the result array
   3. map() abstracts away everything except the mapping part that you need to pass.
   4. Similarly, you can create HOF that return a function (hence are called function factories). The argument you pass to such HOF determines the specific job performed by the function it returns.

## Global Object

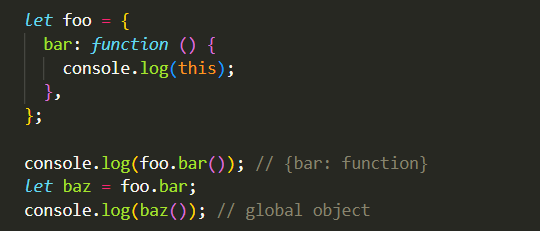
1. JS creates a global object every time it is run. This global object serves as an implicit execution context for function invocations
2. In Node.js global object is an object named *global*, while in the browser, it is the *window* object.
3. It contains important global properties such as:
   1. Values: Infinity, NaN
   2. Functions: isNaN, parseInt, setTimeout

**Note:** Even though you can directly use isNaN() in your code, use Number.isNaN() to avoid unexpected behavior like a string getting coerced to NaN and returning true.

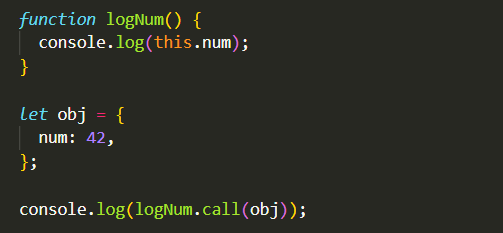
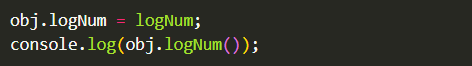
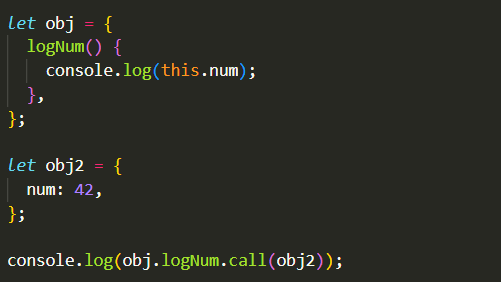
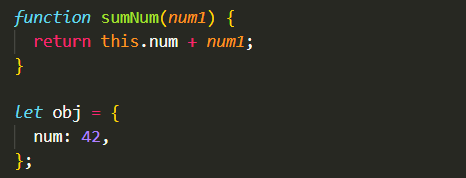
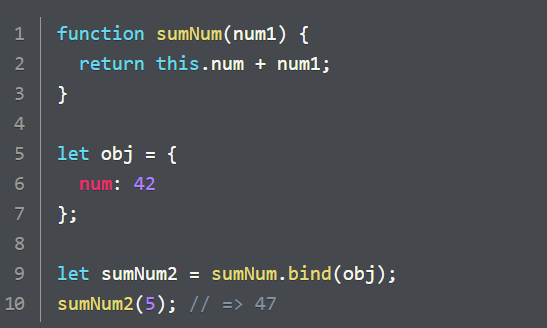
1. Just as with other JS objects, you can add properties to global object as well, ex: global.foo = 1;
2. Such modification of the global object is useful when you need to add certain properties in Node that will be useful in multiple modules.
3. Any undeclared variables (without var, let, const) are automatically added to the global object as a property.

## Implicit and Explicit Execution Context (EC)

Here we discuss this and how JS determines its value in a function/method call.

1. The EC refers to the environment in which a function executes, this environment dictates the value of this. When we are talking about the E.C of a function/method call, we are talking about the value of this when that code executes.
2. How you invoke a fn/method, determines its E.C for that invocation. E.C does not depend on:
   1. How you define the fn/method
   2. Where you define them
   3. When you call them
3. Thus two invocations of the same fn can have different E.C depending on how you make those calls.
4. There are two ways of setting the E.C (context) to a fn/method call
   1. Explicit: The E.C that you set explicitly
   2. Implicit: The E.C that JS implicitly sets when your code does not provide an explicit context.
5. Setting the E.C is also called binding this (or setting the binding), since we are binding this to a specific object when the fn/method is called.
6. **Function E.C (implicit)**
   1. Every JS function call has an E.C and an available this keyword. Every time you call a fn, JS binds this to some object, if no explicit binding is provided by the fn caller, then this gets bound to the global object (window object in browser).
   2. Now, if you use this to access or modify properties, you will access or modify properties on the global object. **Note:** If strict mode is enabled (“use strict;”) then this is implicitly assigned to undefined instead of the global object.
7. **Method E.C:** Implicit and Explicit.
8. **Implicit:**
   1. When you call a method that belongs to an object, the E.C inside the method call is the object used to call the method, this is called Method E.C. This is considered an implicit E.C.
   2. The E.C inside the method call is the object used to call the method. However, this is not always the case, since the context is determined solely by how you call the fn/method. Here foo.bar() is considered a method since we are using the method call syntax to invocate bar, but there are other ways (8.3).  
      ****
   3. If we assign foo.bar to variable baz and then call baz, then baz is assigned to the fn and then being called as a stand-alone function and hence its E.C will be the global object (not the foo object).
9. **Explicit (Method/fn E.C):**
   1. It is possible to provide explicit E.C to any fn/method, and it doesn’t have to be either global object or the object that the fn/method is being called upon. You can use any object (including null) as the E.C for any fn/method.
   2. There are two main methods to do this in JS: **call()** and **apply()**. The general structure of call and apply is:
      1. someObj.someMethod.call(context, arg1, arg2, …..)
      2. someObj.someMethod.apply(context, [arg1, arg2, …..])

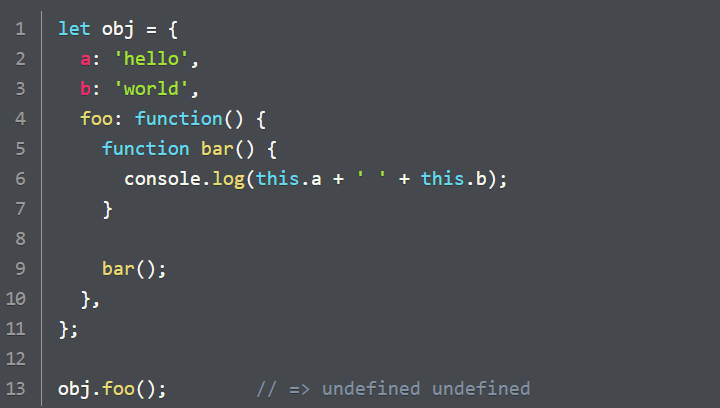
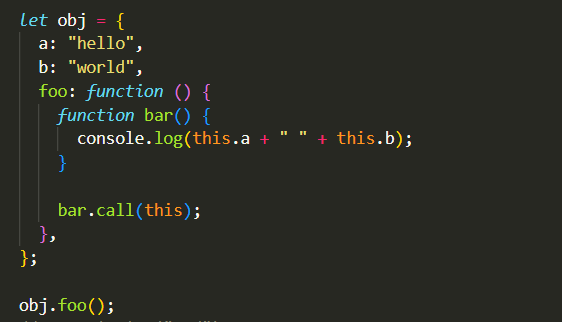
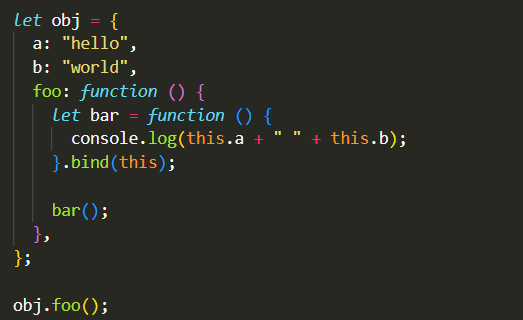
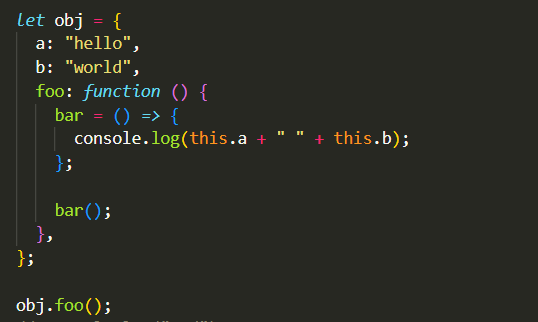
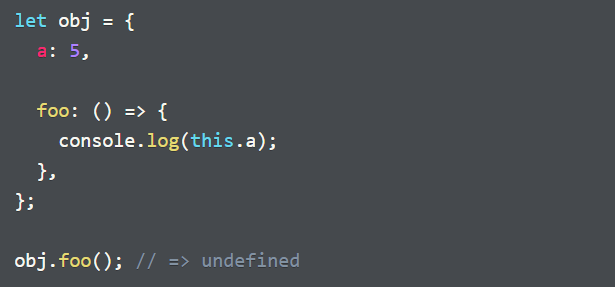
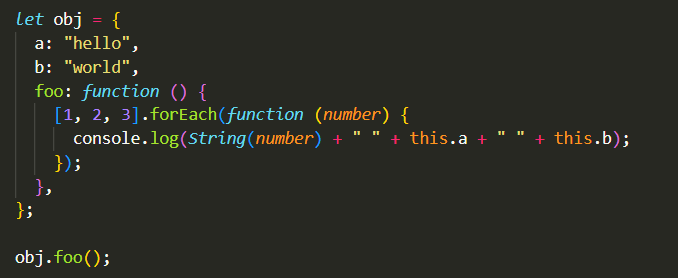
The only difference between call and apply is that in apply we pass an array of arguments, instead of all the arguments individually (as a comma separated list). However, since ES06, if we have an array of arguments we can simply use call() with spread operator.

* 1. Since all fns are objects, and all fns contain the method call, we can use call to execute a fn with an explicit context.  
     
  2. The above code is the same as if the lognum function was added to the obj object, and then invoked from within that object.   
       
     Here we are mutating the obj object, which we wouldn’t need to do if we simply used call().
  3. Along with functions, we can also use call to explicitly set E.C on methods:  
     
  4. What if the function takes in an argument?  
     
  5. Can we use call in such a way that it mutates the obj object’s num property?  
     
  6. **Hard binding functions (with context):** We can use **bind()** to permanently bind a function with a context. Once bound, the function/method cannot be unbound using call, apply or another bind.  
     
  7. Unlike call/apply, the bind function *does not invoke the fn*, but rather return a new function that is permanently bound to the object passed as 1st argument. This function can then be passed around and called without losing its context.

## Dealing with Context Loss

Functions do not lose context but the context they hold might not be the one you are expecting. However, if you understand E.C and closely follow it, you will never be surprised by the value of this.

Let us talk about such scenarios that look like they have lost their context:

1. **Method Copied from Object:** Below, we try to execute the method greetings of john object, inside the function repeatThreeTimes. However this fn scope do not have access to the john object.  
     
   It is possible to pass around a function, such that by the time it gets invoked, the originally bound object may be out of scope.
   1. Solution1: We can make the function repeatThreeTimes accept the context object as a second argument  
        
      Array methods like forEach, map, filter also use this technique, they take callback fn as argument and an optional thisArg context object that can be used as the callback’s E.C.
   2. Solution2: hard bind the method’s context using bind().  
      
      1. Advantage of using bind: Once you bind a context to a function, the binding is permanent and does not need to be repeated no matter how many times it gets called.
      2. Disadvantage: It is no longer possible to determine the context by looking at the invocation of the final function.
2. **Inner (nested) functions loosing the surrounding context:** Here we see how the functions nested within the object methods loose their surrounding context.  
     
   Here (on line 9) bar() is being invoked as a stand-alone function (not as a method) thus its E.C is the global object and not the obj object. Let us see some solutions for preserving the context (obj):
3. **Solution1:** Variable in outer scope  
   If you define a variable: let = this OR self = this in the outer scope of the inner function, you can use the value stored in that variable within your inner function.  
   
4. Solution2: Call the inner function with explicit context  
   We can use call/apply methods to explicitly provide a context when calling the inner function.  
   
5. Solution3: Using bind on the inner function to get a new function whose E.C is permanently bound to the object.  
     
   Here, we are calling bind on the fn expression which is returning a new fn and storing it in the bar variable. Instead, we can also use a function declaration but then we will need an extra step to store the newly returned function.
6. Solution4: Using an arrow function  
   Arrow fns do not have an E.C of its own, instead it uses E.C from its surrounding context, in which it is defined.   
     
   Hence arrow fn defined inside another fn always has the same context (that at the time of definition). Using arrow fn is similar to using bind() in this way, that is you do not need to worry about loosing the surrounding context. The arrow fn once created always has the same E.C as the fn that surrounded it when it was created.
7. **Arrow fns should not be used to define object methods:**The reason for this is that since arrow fns always get their E.C from the surrounding context, if there is no surrounding fn, it will instead take the global context (this = global object).   
     
   This is because, arrow fn takes its context from the immediate let statement, which in the above code is at the top-level (global scope).  
   **Note:** Here it does not matter how we are invoking the fn, we are calling obj.foo via the object and hence that E.C must be used, but that is not happening. This discrepancy is why you must never use arrow fns to write object methods.
8. **Context-loss 3: When passing a function as an argument to another function**:  
     
   Passing a function as an argument to another function, strips it of its E.C which means the function argument gets invoked with the global context. This is similar to copying a method from an object and using it as a bare function.
9. Solutions:
   1. Using a variable in outer scope
   2. Using bind() method on the callback fn
   3. Using an arrow fn as callback fn
   4. Using thisArg argument of the method to define the context (works in map, some, every etc.)

# Lesson3: Object Creation Patterns

Even though JS has ‘class’ keyword, it does not implement behavior sharing using class-based inheritance. Instead, it uses the object prototype to share properties. This is imp. in understanding how JS creates individual objects as it forms the basis of all object creation patterns in JS that feature behavior sharing.

## **1 Factory functions**

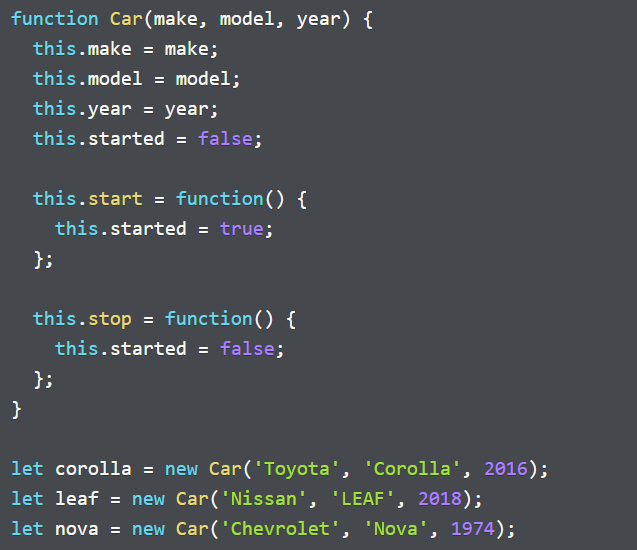
1. Also called Factory Object Creation Pattern, they provide a simple way to create related objects based on a predefined template.

## 2 Object Orientation

1. OOP is a pattern that uses objects as the basic building blocks of a program instead of local variables and functions
2. OOP approach puts data and procedures that manipulate the data together into one container (objects).
3. We no longer deal with primitives or composites of primitives but smart objects that can perform actions on the data they themselves own. This way we move the complexity inside the object instead of exposing it globally.
4. When we make changes we restrict those changes to the object, they don’t ripple throughout the project. Thus, we make maintenance easier by limiting the scope of changes.
5. Since we can build as many instances as we want from a given template/object, we reduce duplication.
6. OOP code is much easier to understand as the relation between code and the data is evident. OOP code becomes more useful as the size of codebase increases.
7. OOP makes the following questions easier to answer:
   1. What are the important concepts in a program
   2. What are the properties and operations of various objects
   3. How to create the respective objects
   4. Where should we add new properties and methods for an object

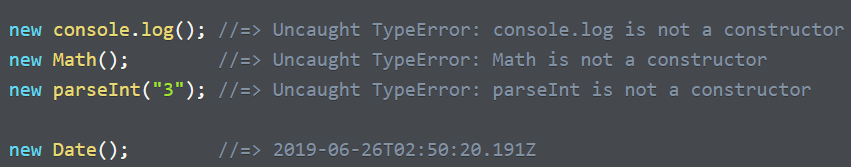
## 3 Constructors

1. Also called Object constructors, are another way of creating objects in JS.
2. Defining a constructor is almost same as creating a function, with a few differences:
   1. Function parameters match with associated properties (a property can also be computed from other properties or retrieved from a database)
   2. We call the constructor with the new keyword.
   3. We use this to set the object’s properties and methods.
   4. We don’t supply an explicit return value.

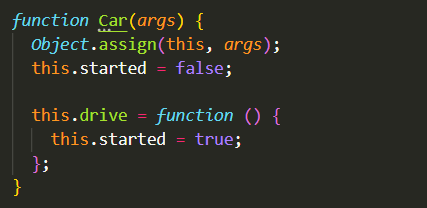
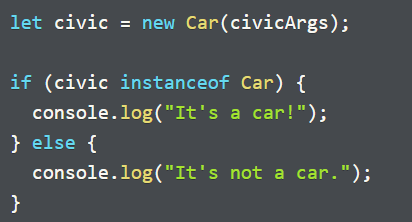


1. We already know that this refers to an object, but which object does it refer to here? Its value depends on how we call the function.
2. The combination of using new with function call is what treats it as a constructor. But what is so special about calling a function with a new keyword? When you invoke a function with new:
   1. It creates an entirely new object.
   2. It sets the prototype of the new object to that of the object referenced by the prototype property of the constructor function (Ex: Car.prototype).
   3. It sets the value of this inside the constructor function to that of the new object.
   4. It invokes the function and since this refers to the new object, we use it within the function to set properties and methods for the new object.
   5. Finally, once the function/constructor finishes running, new returns the new object, even if the function did not explicitly return anything.

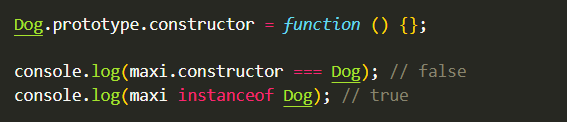
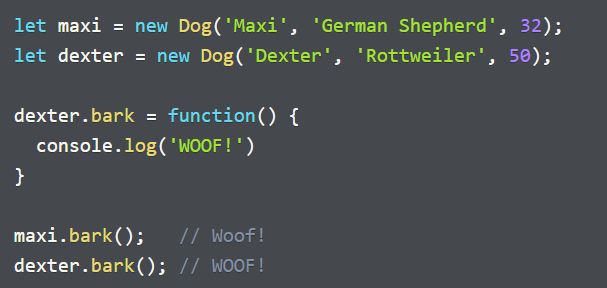
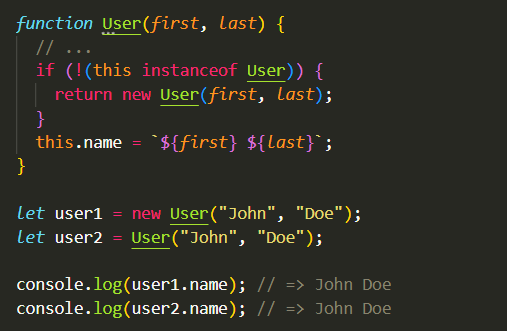
Note: If we invoke the fn without new, it works as an ordinary fn, no new objects are created, so new won’t point to a new object.

1. **Who can be a constructor?** : Almost any JS function can become a constructor with a new keyword except:
   1. You cannot call an arrow function with new, since arrow function does not have a prototype property.
   2. You can use new on methods that you define inside objects. However, you cannot use it on methods that are defined using the concise method syntax.
   3. Many built-in methods or objects are incompatible with new  
      
   4. new is also incompatible with special functions called generators.
2. Using new to call a function that returns an explicit value:
   1. return value is a primitive: Makes no difference. It still returns the main object.
   2. return value is another object: It returns the returned value instead.

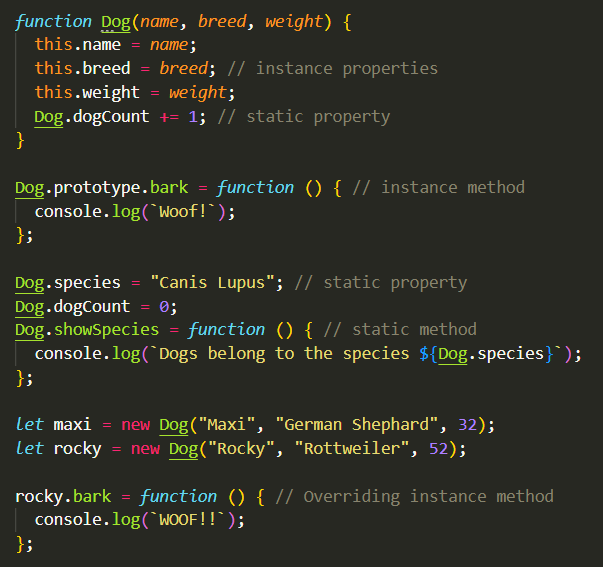
In short, the constructor ignores primitive return values.

1. Supplying constructor arguments with plain objects: Sometimes a constructor can have too many properties, which can be a source of error. In such a case an object containing the properties is often passed as argument to the constructor.  
     
   Here, args is an object containing all the arguments.
2. We can use Object.assign() to simplify this further.  
   
3. **Determining the type of an object:** The instanceof operator lets us determine, whether a given instance has been created by a specific constructor or not. The object returned by the new operator contains some info that ties it back to the constructor that created it. The instanceof operator uses that info to determine the constructor that created the object.  
   
4. Remember, when you call a function with new, its implicit execution context is the new object.

## Constructors with prototypes

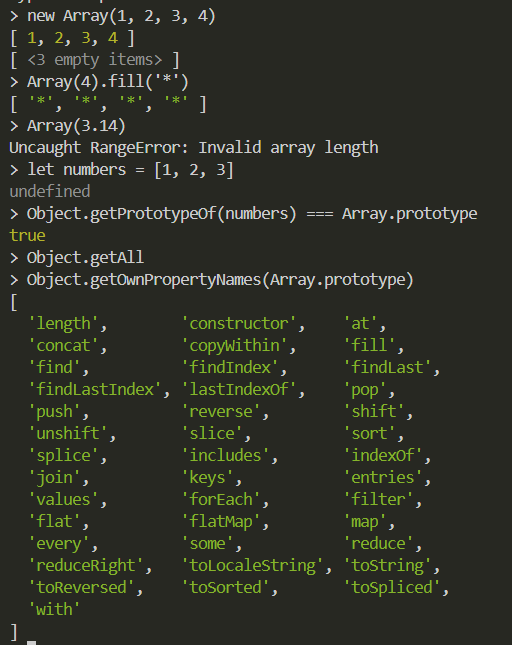
1. Each time we create an instance object from a constructor, it copies all the properties and methods in its new object. The properties of each instance are different but methods are still the same, hence this is inefficient and wasteful from memory use standpoint.
2. **Solution**: We can delegate the methods to the prototype of the constructor. All the instances can then access the methods from within this prototype object (using the prototype chain):
   1. All constructors (and functions) have a prototype property that we call function prototype. When you call a function *Car* with *new* keyword, JS sets the new object’s prototype to the current value of Car’s prototype. Thus, the constructor creates an instance whose prototype references Car.prototype.
   2. Since arrow fns do not have prototype property, you cannot call an arrow function with the new keyword.
   3. Note, that the constructor doesn’t have to explicitly set the prototype of instance using this to Dog.prototype, JS does this for us when we call the constructor using *new*.
   4. When we use this in bark(), it does not refer to the prototype object (see code above), but rather to the instance. Why is that?
   5. When you call a method on an object, JS binds this to the object whose method it was called on. If it doesn’t find the method in the object, it looks for the method in its prototype, but this does not change the binding of this (the fn whose method it was called on).
3. The constructor property: It is a property of the prototype object (Dog.prototype). It helps in determining the type of object (just like *instanceof*).  
   
4. Be careful though, it is also possible to reassign the constructor property to something else. However, the *instanceof* operator still works.  
   
5. Overriding the prototype: Just because all instances share methods in the prototype, does not mean they cannot have their own distinct methods. You can override the method in prototype by assigning a method to the instance object.  
     
   The dexter object now has its own bark() method that overrides the bark method in Dog.prototype.
6. **Scope-safe constructors:** Constructors that can be used regardless of whether they were called using the new operator or not. Here User can be used both with and without new, by the user.  
   Most JS built-in constructs ex: *Object*, *RegExp*, *Array* are scope-safe, *String* is not

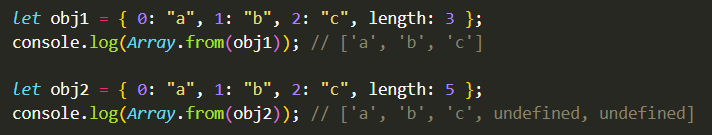
## Instance properties and methods, Static properties and methods

1. Individual objects of a specific data types are called ***instances****.*
2. **Instance properties**: Properties that belong to specific instances of some type. Ex: name, breed and weight properties in our Dog type.
3. **Instance methods**: functions that are also properties of an object/instance. They can be stored directly on instance objects but are mostly stored in the prototype object (they still operate on individual instances though).  
   Any method defined in any part of the prototype chain of an object, is considered an instance method of the object.  
   
4. **Static properties**: Defined and accessed directly on the constructor (type), not on an instance or the prototype object.
   1. They are also directly defined on the type.
   2. Commonly used to keep track of all the objects created by the constructor.
5. **Static methods:** functions defined directly on the type. Examples of static methods on built-in JS constructors: Object.asssign, Date.now, Array.isArray.

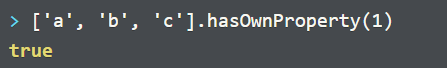
## Built-in Constructors

* The Array constructor:   
  It is a scope-safe constructor but it is recommended to be used with ‘new’.



1. The **Object.getOwnPropertyNames()** above give all the instance methods of Array type.
2. **Array.isArray()** lets us distinguish between arrays and other types of objects (typeof doesn’t help).
3. **Array.from()**: takes an array like object as an argument and returns a new array with equivalent array values.
   1. **Array like object**: Any object that has a length property and provides indexed access to some of its properties (using index-notation)  
      
   2. **Use case:** Some fns/methods returnobjects that resemble arrays in some ways but serve some other purpose. Ex: DOM returning a node-list (Arraylike object) when you request a list of elements from the browser. Such objects can store live-dynamic data that can change/update itself. An array wouldn’t work for such an application but you can still convert this to an array using Array.from().

### Object constructor

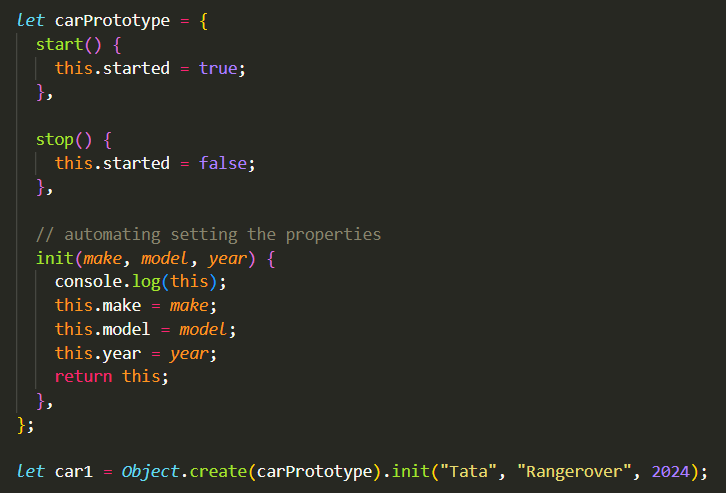
1. You can use either object constructor or an object literal syntax to build an object: **new Object()** or **{}**
2. All objects inherit from Object.prototype and hence have access to all the instance methods defined in Object.prototype ex: Object.prototype.hasOwnProperty(), Object.prototype.isProtypeOf() etc.
3. Since arrays are also objects, they also have access to these properties.  
     
   We can think of integer indices as properties of an array.
4. Infact, almost all JS objects inherit from Object.prototype, either directly or further down the prototype chain. As we know, it is possible to create objects that do not inherit from Object.prototype (bare/clean objects).
5. **READ FURTHER IN LAUNCH SCHOOL**

## Classes

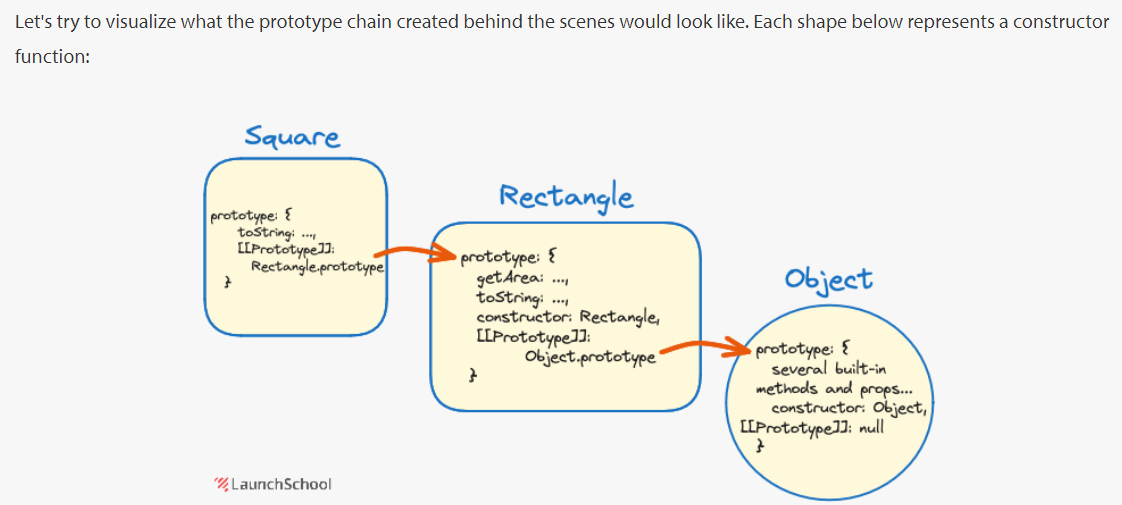
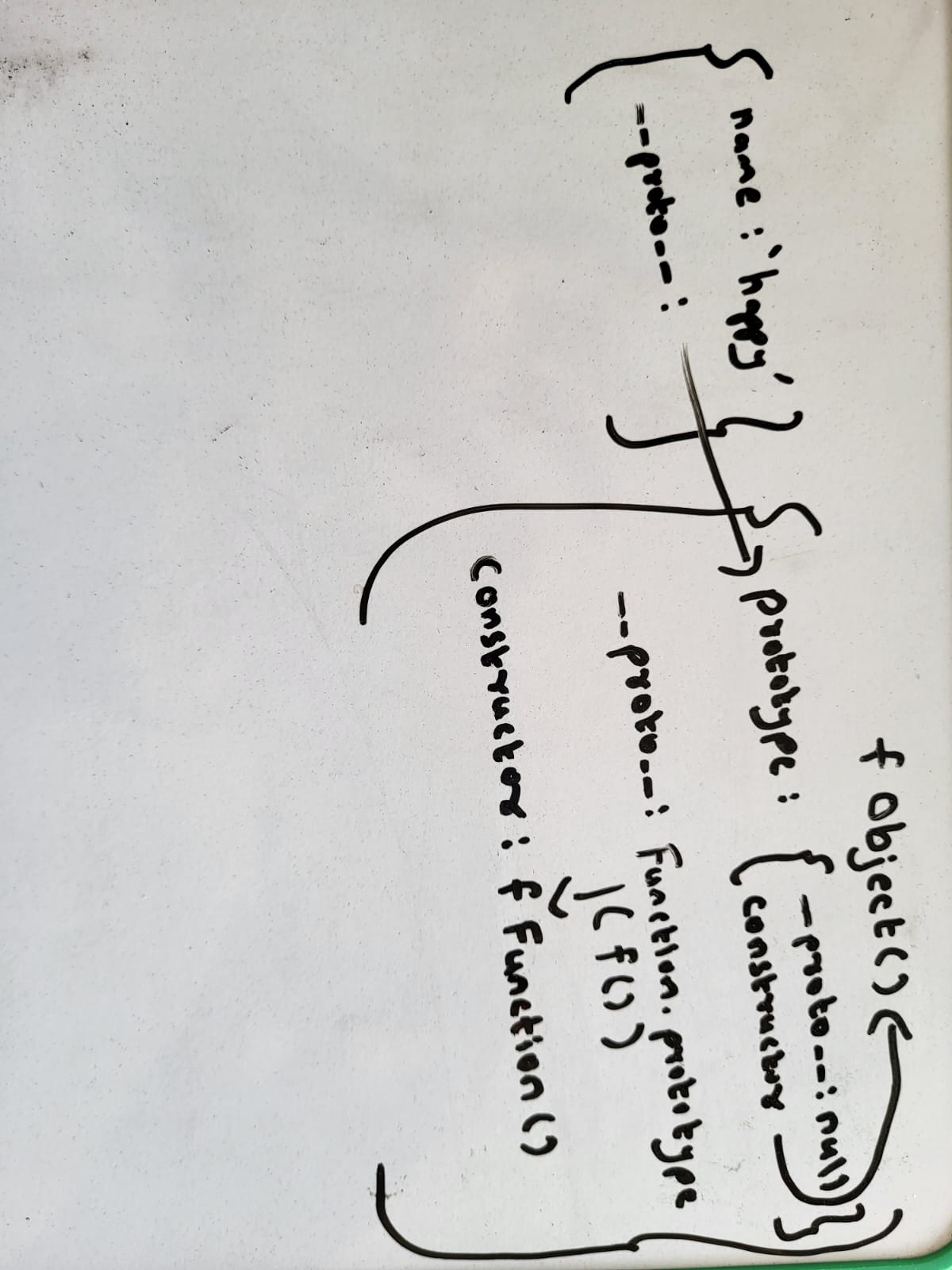
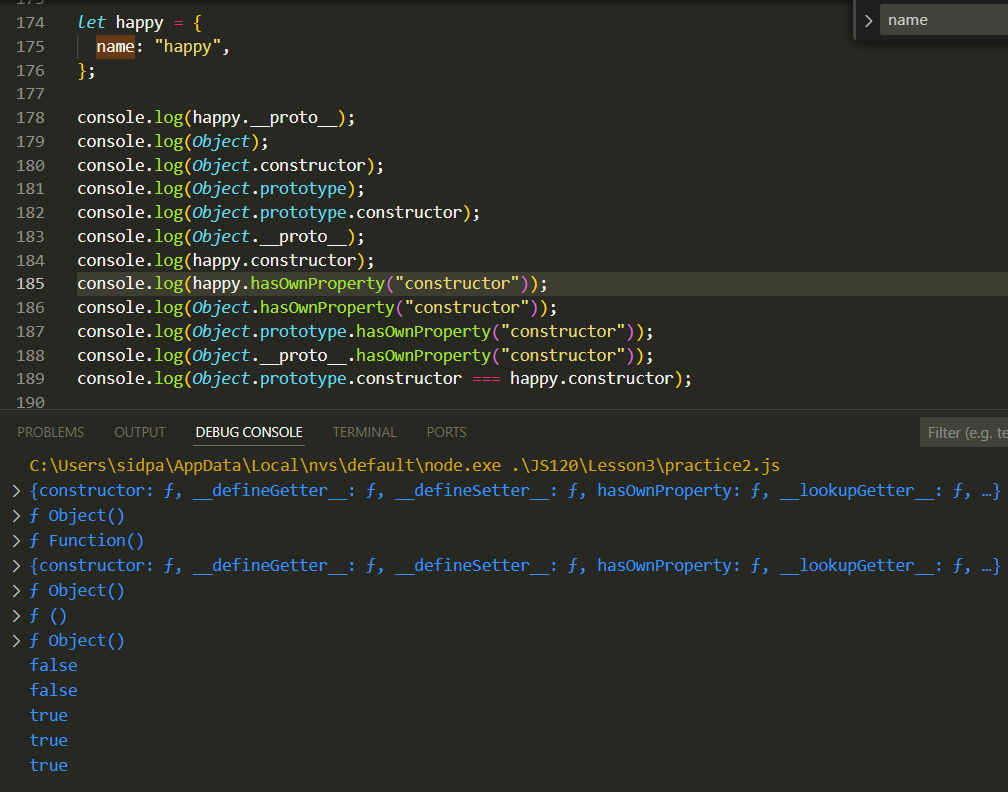
1. Cleaner, more compact alternative to constructors and prototypes. Like functions, they are first-class citizens and come in the form of declarations and expressions.
2. Sometimes referred to as ES06 classes, to differentiate them from classical classes available in other OO languages.

# Lesson4: Subclassing and Code Reuse Patterns

## OLOO pattern

1. Objects Linking to Other Objects (OLOO) is a pattern where we make a prototype and add to it all the properties/methods common to all the instances that we create.  
   
2. We make the init() return this so that we can chain object creation with properties invocation (using init()).
3. **Advantages** of OLOO: Although we can use both factory functions and OLOO to bulk create objects of the same type, using OLOO provides the advantage of **memory efficiency** as all objects created inherit methods from a single prototype object.
4. **Disadvantage**: OLOO pattern **needs a method that initializes** the object after it is created (init()).
5. **Note**: OLOO pattern is not widely used and should be only used if your team is already working on it.

## Subtyping with constructors and prototypes

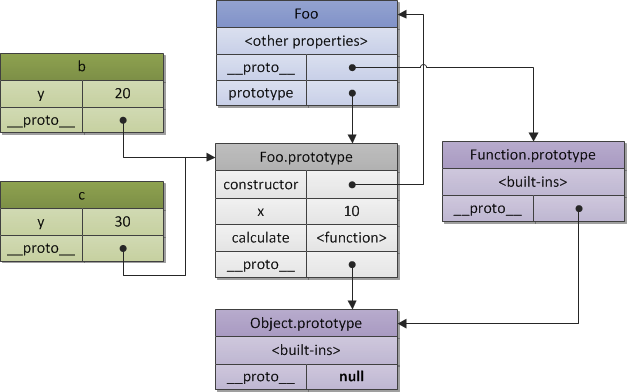
1. Implementing inheritance using Constructors and prototypes (by linking prototypes, using Object.create).  
     
   
2. Such **prototypal inheritance/delegation** is also called **object inheritance**, since it works with one object at a time., the object’s internal [[Prototype]] property points to another object and the object can delegate method calls to that other object.
3. We can also use the class syntax to create such pseudo-classical object construction (and inheritance). Here we use the extends keyword in class declaration to create a class that is a child of another class. We also use super() to call the constructor of the super class.  
   **Note**: When you call super() in a subclass’s constructor, you must call it before you use this in that constructor.
4. Constructor property in JS:  
   super;
5. Difference between \_\_proto\_\_ and prototype:
   1. \_\_proto\_\_ is the actual object which is used in the lookup chain to resolve methods, etc. prototype is the object used to build \_\_proto\_\_ when you create an object with new.

(new Foo).\_\_proto\_\_ === Foo.prototype;

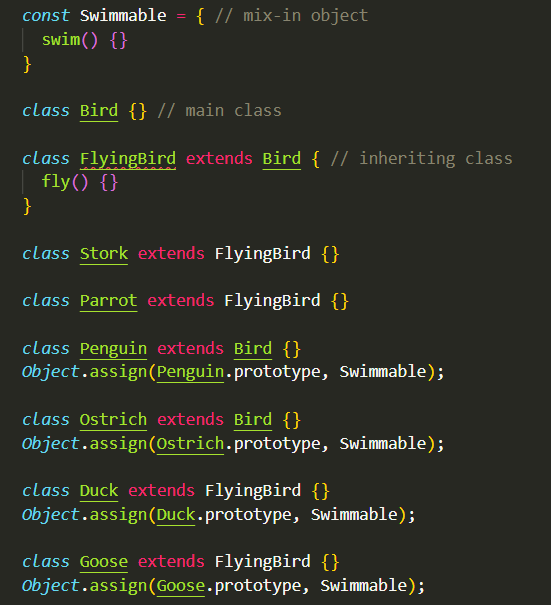
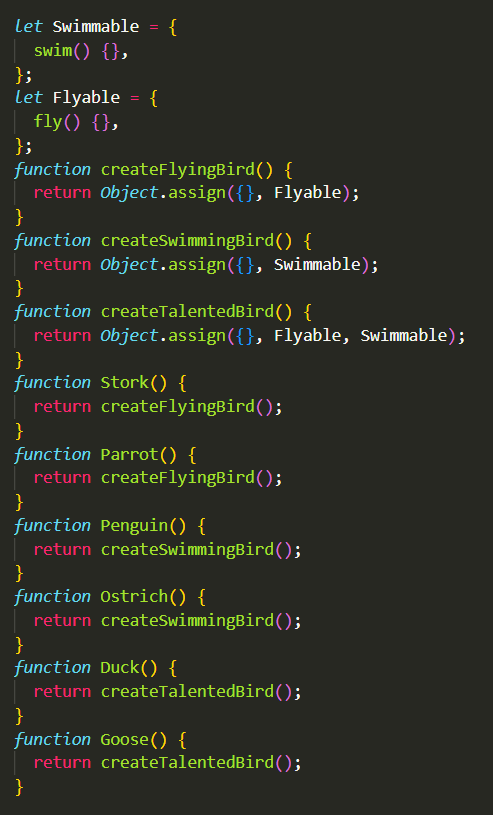
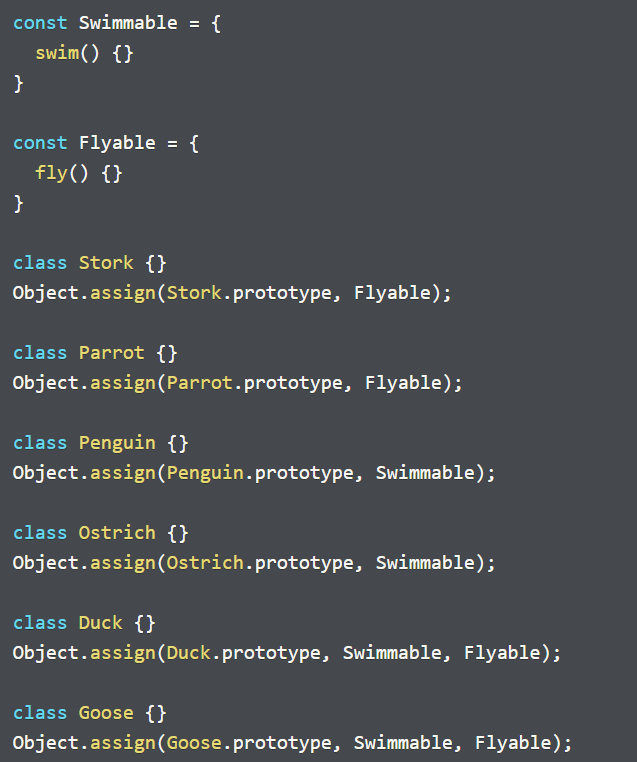
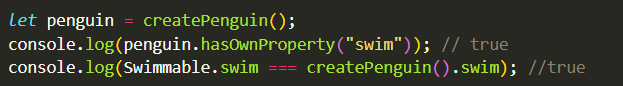
(new Foo).prototype === undefined

* 1. Prototype is only available on functions because it is derived from function objects, but in anything else it is not. However, \_\_proto\_\_ is available everywhere.
  2. Every object has a prototype, a constructor function has a \_\_proto\_\_ which references the Function.prototype (Stores built-in function methods). This Function.prototype in turn references the Object.prototype via its \_\_proto\_\_ property.
  3. The prototype property of the constructor (Foo.prototype) stores its constructor that references back to the function object itself, thus enabling to write code like:

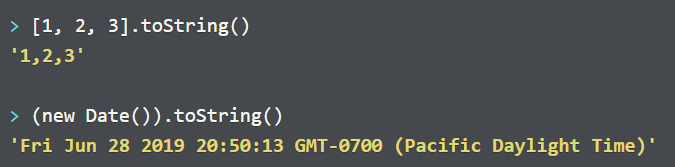
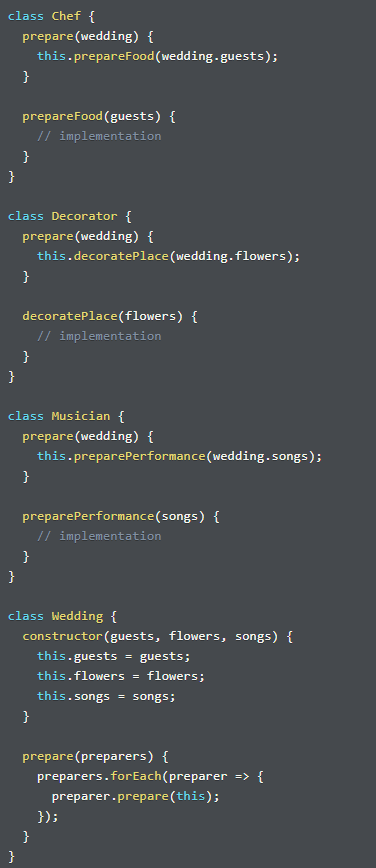
Foo.prototype.constructor = Foo;

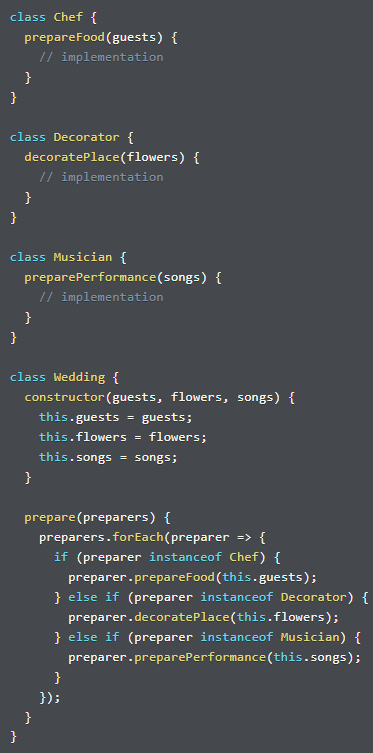
* 1. An instance however, does not have a prototype property and its \_\_proto\_\_ property references the constructor function’s prototype (Foo.prototype).  
     

## Code reuse with Mixins

1. A problem in JS with inheritance is that an object can inherit from only 1 object and classes can extend only 1 another class. Hence only single inheritance can be implemented, not multiple inheritance.
2. A compromised solution to this is using **mix-ins**. It is a pattern that adds methods and properties from one object to another. This is not delegation with prototypes, mix-ins merely copy properties from one object to another using Object.assign or other similar technique.
3. An example of using a mix-in with classes. A mix-in, is an object that defines one or methods that can be “mixed-in” to a class. This grants that class access to all of the methods in the mix-in object. This is the only workaround for the lack of multiple-inheritance.
4. Pattern-1: Below, we have created a Swimmable object (mix-in) with the swim() method to mix it into our various swimming birds. We have used Object.assign() to add the methods from Swimmable to the prototype objects of those classes.  
   
5. Pattern2: We can also use mix-ins for both SwimmingBird (Swimmable) and FlyingBird (Flyable):  
   
6. Pattern3: **Using mix-ins with factory functions**-  
   Some developers suggest that inheritance fails at modeling some scenarios, but using mix-ins with factory functions can model any object relationships. Hence, they can be used exclusively for all OOP without bothering with class/constructor at all.
7. Downsides of using factory-functions:
   1. Every new object created will have a new copy of all the methods, outside of the ones we share with mix-ins. This can be taxing on memory-resources (more than the memory requirements of mix-ins).
   2. You cannot determine the type of objects created with a factory function (instanceof operator would only return the generic type object)
8. **Note**: We are using Object.assign() to add all the methods from the mixin to ‘{}’. Since functions are objects, and objects are passed by reference (not a copy), the new properties created have the same memory address as the mixins property reference, i.e.:  
   

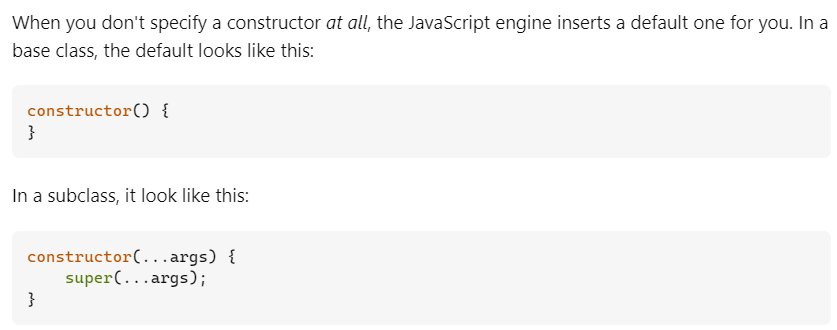
## Polymorphism

1. It is the ability of objects with different types to call the same method (in name) invocation and exhibit different behavior. That is, data of different types can respond to a common interface in their own unique ways. This can lead to a more maintainable code and hence is a crucial concept in OOP.
2. There are two cheap ways to implement polymorphism:
   1. Polymorphism through inheritance
   2. Polymorphism through duck typing
3. Polymorphism through inheritance:  
   
   1. Every object in the animals array is of different type but the code uses each of those objects without caring about their type, the only assumption being that they all have move() method. The interface of this class hierarchy lets us work with all these types in the similar way, even though the implementation for each may be dramatically different.
   2. Sponge and Coral do not have their own move() method, hence they inherit it from their superclass Animal which does nothing here. This is polymorphism through inheritance.
   3. Fish and Cat have their own move() methods hence they respond differently by simply overriding that method from their super class. Overriding is treated as an aspect of inheritance, hence it is still treated as polymorphism through inheritance.
   4. toString() method in JS is an example of polymorphism through inheritance. The Object type provides a default implementation to toString() that other types inherit but it only returns a string [Object Object] when called on an object. However, other types can override this method to return a customized string representation.  
      
4. Polymorphism through Duck Typing: It is when object of different types use the same method name to perform different but related functions
   1. The term ‘duck typing’ refers to the concept that If something looks like a duck and quacks like a duck then it is indeed a duck. Similarly, if the objects involved uses the same method name and takes the same arguments, then we can treat the object as belonging to a specific category of objects.
   2. Ex: An application may have a variety of methods that respond to a mouse click by calling a method like say handleClick. Those elements may be completely different, ex: checkbox vs a text input field even though they are all clickable.
   3. Duck typing is an informal way to ascribe type to objects. Classes and constructors provide a formal way to do that.
   4. Example of OOP with and without duck typing:



The R.H.S implements duck-typing. Notice that even though there is no inheritance, each preparer type, provides its own prepare method that takes in its own argument and outputs its own thing. If later we want to add more classes with their own prepare method, its much easier to scale with duck typing.

## More Insights

1. What happens if you don’t specify a constructor in a subclass, Ex: class Cat extends Animal {} ?  
     
   As you can see, in a subclass the default constructor passes along all of its arguments to the superclass constructor. You need to specify a constructor in a subclass only if it does something different or extra.
2. **Mixins vs. Inheritance**: Mixins are more appropriate in a has-a relationship (Where you want some additional functionality) while inheritance is more appropriate in a is-a relationship (Where you want to extend the abilities of a class).

## Tic-Tac-Toe

### Description

1. Tic-Tac-Toe is a 2-player board game
2. The board is a 3 x 3 grid
3. Players take turn marking a square with a marker that identifies them.
4. Traditionally, the player that goes 1st uses the X marker and the player thereafter uses the O marker.
5. The first player to mark 3 tiles in a row wins the game.
6. A row can be horizontal, vertical or any of the two diagonals (top-left to bottom-right or top-right to bottom-left).
7. There is one human player and one computer player.
8. The human player always moves (places a marker) first, at least in the initial version of the game, you can change that later.
9. **nouns:** game, board, square, grid, marker, row, player, human, computer
10. **verbs:** play, mark, place (check winner, display winner)

### Organizing the significant nouns and verbs to show relationships between them:

1. Game
2. board
3. row
4. square
5. marker
6. player
   1. mark
   2. play
   3. human
   4. computer
7. Every player: name, health, strength, and intelligence
8. Each character receives, use rollDice() method:
   1. a health value of 100.
   2. a random strength value (between 2 and 12, inclusive).
   3. a random intelligence value (between 2 and 12 inclusive).